Enhanced Graph Rewriting Systems for Performance Aware Dynamic Software Architectures
Cédric Eichler, Thierry Monteil, Patricia Stolf, Alfredo Grieco, Khalil Drira

To cite this version:
Cédric Eichler, Thierry Monteil, Patricia Stolf, Alfredo Grieco, Khalil Drira. Enhanced Graph Rewriting Systems for Performance Aware Dynamic Software Architectures. Software and Systems Modeling, Springer Verlag, 2014, <10.1007/s10270-014-0433-1>. <hal-01228244>

HAL Id: hal-01228244
https://hal.archives-ouvertes.fr/hal-01228244
Submitted on 12 Nov 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Enhanced Graph Rewriting Systems for Performance Aware Dynamic Software Architectures

Cédric Eichler · Thierry Monteil · Patricia Stolf · Alfredo Grieco · Khalil Drira

Received: date / Accepted: date

Abstract Methodologies for correct by construction reconstructions can efficiently solve consistency issues in dynamic software architectures. At the same time, they may be unfit to characterize a system from a performance perspective. This stems from efficiency and applicability limitations in handling time-varying characteristics and related intra-dependencies. In order to lift these restrictions, an extension to graph rewriting systems is proposed herein. Noticeably, the resulting formalism integrates changing constraints that reflect the adequacy of a configuration in accordance to potentially concurrent secondary objectives (e.g., quality of service, energy consumption, and robustness). The suitability of this approach, as well as the restraints of currently available ones, are illustrated and analysed with reference to a concrete use case. This last, known as DIET, consists in a load balancing application for dispatching computational jobs in a distributed infrastructure, which is part of SysFera-DS, a broader industrial solution for federating and managing hybrid HPC environment. This investigation demonstrates that the conceived solution can: (i) express any kind of algebraic dependencies between evolving requirements and properties; (ii) ease the manipulation of attributes of the system and its elements with respect to classic methodologies; (iii) provide an efficient access to attribute values; (iv) be fruitfully exploited in software management systems; (v) guarantee theoretical properties of a grammar, like its termination.

Keywords Dynamic software architectures · Software performance · Software correctness · Graph rewriting systems · Constrained and attributed rewriting systems.

1 Introduction

Dynamic software architectures enable adaptation in evolving distributed systems [13, 22]. Their description cannot be limited to a unique static topology, but it has to encompass the entire scope of possible configurations [19]. This scope is characterized by an architectural style, qualifying what is correct and what is not. Once this distinction made, system transformations themselves must be specified to depict their applicability conditions and effects. A crucial undesirable implication of these evolutions is a potential loss of correctness, the system withdrawing from the scope of consistency.

Besides correctness, the system has evolving requirements and secondary objectives, which are tightly linked to the appropriateness or efficiency of the architecture. For example, configurations can be evaluated with reference to quality of service, energy consumption, and robustness to software or machine breakdowns. These objectives are potentially concurrent. In fact, deploying more software components or using more machines may ameliorate robustness but worsen energy consumption. Also, the satisfaction of an objective depends on the properties of each software component, such as the machine it is deployed on, and the components reachable from it. In turn, those characteristics are dynamic and may be interdependent. As a matter of fact, the set of entities accessible through a component of the system...
hence, modelling a system to ease its management carries two particular aspects which are usually considered separately [37]: correctness and performance. These concerns motivate the need for suitable description languages and *formalisms avoiding ambiguities for correct architectural design, management and analysis*.

Formal unambiguous methods are necessary to study the consistency of a system at a given time, i.e., its compliance to an architectural style. Several ways of doing so have been developed in the literature. The most immediate approach, checking the consistency of the system at run-time, may lead to combinatorial explosions and the necessity of roll-backs if it is discovered that the system is in an inconsistent state. To efficiently tackle correctness in the scope of dynamic reconfiguration, correctness by construction [34] through formal approaches have emerged [4, 15, 6]. *Based on formal proofs and reasoning in design-time, they guarantee the correctness of a system, requiring little or no verifications in run-time*. A way to achieve such proofs is to investigate the properties of transformations with regard to consistency preservation, so as to ensure that if a transformation is applicable on a correct configuration its result is another correct configuration.

Modelling dynamic systems with graph-based methodologies has a long tradition [25, 26, 16, 5, 31, 12]. As generic models, graphs may be used to represent a broad range of systems according to diverse architectural views. Graph rewriting techniques allow to elaborate style-based frameworks for the specification of dynamic systems granting correct by construction, style-preserving, evolutions. However, they exhibit restraints critically weakening the possibility of assessing a configuration appropriateness when considering non-simplistic systems.

With reference to a real use case, this article first highlights limitations of currently available graph based methods in describing system properties and their inter dependencies. A formal extension of graph rewriting systems is then proposed to lift these shortcomings. The pivotal features of this new formalism are: *mutators, admissible relationships* specification, and *constraint oriented* encoding. In fact, inspired by classical string rewriting theory [30], *mutators* are introduced within graph rewriting rules as a lightweight approach to *attribute modifications*. On the other hand, *attribute interdependencies* are expressed through *algebraic operators* that allow to *characterize admissible relationships*. Finally, to ease application management operations, the appropriateness of a configuration in accordance to secondary objectives is reflected by constraints. To this regard, it is worth remarking that a characteristic may be unknown, due to a lack of information or the postponing of a decision. Therefore, the solution proposed hereby integrates both soft and hard dynamic constraints as expressions of a ternary logic system.

The investigated use case, known as *DIET* [8], consists in a hierarchical load balancer for dispatching jobs over a distributed infrastructure. This application is part of *SysFera-DS* , an industrial solution for federating and managing hybrid HPC environment. Its architecture, represented adopting a component based view, is composed by a set of agents: a master agent manages pools of service providers through none, one or several strata of layer agents. Master and layers agents both have a minimum and a maximum number of managed entities. Each agent is registered to a naming service driving their communications. In this context, the system *correctness* is related to:

- the tree structure of its agents, whose root and leaves are respectively a master agent and service carriers, each other elements being layer agents.
- the number of agents the master and each layer agents manage.
- the registration of each agent to the naming service.

Based on this use case, it is demonstrated that the proposed solution brings three main beneficial advantages with respect to classic graph rewriting approaches.

First, characteristics of the system can be more efficiently assessed by combining evaluation on demand and/or update on modification. As usual, a property can be evaluated whenever its value has to be known. However, one may want to avoid frequent evaluations by keeping track of the current value of an attribute and updating it whenever it changes. The choice between these two options rely on the relatives complexities and frequencies of updates and evaluations.

Second, the model allows to quickly grasp the appropriateness of a configuration, identify objectives that can be ameliorated, and component implying constraints violation. Therefore the management of the system and its evolutions is facilitated.

Third, reasoning about theoretical characteristics of the model is eased. For example, we demonstrate that it is simple to guarantee the termination of a grammar, i.e. the fact that there can not be an infinite sequence of its production rules. This property ensure that the set of instances of the

---

1 Distributed Interactive Engineering Toolbox
2 Sources and further information are available at http://graal.ens-lyon.fr/DIET
3 http://www.sysfera.com/sysfera-ds.html
grammar is finite. Hence, its exploration or the construction of any instance can be represented by a terminating algorithm.

The rest of the paper is articulated as follows: existing approaches and their main features are illustrated in the next section. The DIET use case is presented in Sec. 3. Section 4 introduces the proposed formal extension of classical graphs and graph-grammars related theory. Section 5 exploits this new model to characterize the use case, and demonstrates its fitness for appropriateness evaluation and performance-aware management. Finally, Sec. 6 is dedicated to conclusion and outlooks.

2 Related Works

2.1 Language-Based Approaches

Architecture Description Languages (ADL) [29, 2, 14, 28] have been widely used to model software systems [27, 32, 38]. Thanks to a rigorous syntax and semantic, they allow the definition of architectural entities and relations, as well as, the description of the structural and behavioral properties and constraints of a system. However, such languages usually focus on the description instances of an architecture, whereas dynamic aspects have been mildly studied [20]. Darwin [28] and ACME [14] only allow component replication and optional components/connections, respectively. Dynamic-Wright [3] adds evolving capabilities to the language Wright [2], limiting itself to predefined dynamics, meaning that the system should have a finite number of configurations and reconfiguration policies known in advance.

2.2 Model-Based Approaches

General-purpose modelling techniques may provide great means for handling dynamism, thanks to the definition of reconfiguration rules that handle the evolution on an application in run-time. They furnish very intuitive and visual formal or semi-formal description of structural properties [7]. Designing and describing software models using UML, for example, is a common practice in the software industry, providing a standardized definition of system structure and terminology, as well as facilitating a more consistent and broader understanding of software architecture [35]. Nevertheless, the generic fitness of model-based approaches implies some limitations in describing specific issues like behavioural properties. Therefore, they often require the adoption of ad hoc description languages [36, 38] to map architectural concepts into the visual notation of a model (e.g., UML) [26, 21]. Moreover, in spite of their wide acceptance, UML-based descriptions appear to lack formal tools for efficiently guaranteeing consistency, due to the inherent semi-formalness of UML, so that fitter models should be considered.

2.2.1 Graph-Based Approaches

Among model-based approaches, graph-based methods are appropriate for conceiving correct by construction frameworks. Graphs and graphs rewriting have been successfully applied for modelling structural constraints and properties of a vast range of systems in multiple fields, including software architectures. As a generic model, graphs may be used to represent different architectural views, be it component-based [12], service-based [5], event-oriented, or even human applications [31]. Furthermore, this genericness allows, similarly to approaches combining languages- and model-based solutions, the use of graphs to conduct adaptation in systems described with UML.

Within graph-based approaches, a configuration is represented by a graph and graph rewriting rules can express horizontal or vertical transformations, i.e. reconfigurations or refinements. Architectural styles can be characterized by either a type graph [39, 5] or a graph grammar [17, 16]. The first suffers from the same lack of expressiveness as UML-based methods. Graph grammars offer a generative definition of the scope of correctness, where graph rewriting rules have two distinct values. They intervene in both the characterisation of an architectural style as part of a rewriting system and in the specification of consistency preserving reconfiguration rules [16]. This fitness for designing correct by construction transformations is a key motivation for the adoption of graph grammars as a modelling tool of dynamic software architectures.

2.2.2 Attributed Graphs

The very first thing to consider with graph-based models is the definition of attributes, representing the basic properties of a system element. Variable attributes are usually considered in graph rewriting rules alone. However, it may occur in real systems that the value of an attribute is unknown, due to a lack of information or the postponing of a decision. Consequently, attributes of the conceptual graph modelling the system state at a given time can be also variable. Two approaches prevail for the definition of attributes. The most complex is to consider attributes as special vertexes of the graph [11]. In particular, their domain of definition and operations are defined in the form of a many sorted algebraic signatures [10] SIG, thus viewing attributes as elements of
a SIG-algebra [11]. A direct implication is a natural manipulation of attributes using predefined operators and their addition or deletion as regular vertices of the graph. This modularity does not come without drawbacks. Graph rewriting rules rely on finding graph morphisms, an NP-complete problem. As a consequence, it first seems inefficient to increase the size of the input graphs. Secondly, multiplying the application of graph rewriting rules to modify attributes should be avoided. In particular whenever a domino effect is implied, i.e., when changing the value of an attribute recursively impact a chain of interdependent attributes. Indeed, a simpler solution can be adopted where each elements of the graph, i.e., vertices and edges, is assigned a list of couples representing attributes along with theirs domains of definition [31]. The main drawback is that it does not allow to conduct any operation on said attributes. To this end, a novel formalism in presented in the sequel of the paper that mitigates these restraints and make graph rewriting systems also able to cope with software appropriateness.

3 Illustrative Use Case and Problem Statement

3.1 Distributed Interactive Engineering Toolbox

In order to clarify the issues addressed in this article, a practical use case is taken from SysFera-DS, an industrial solution for federating and managing hybrid HPC environment. DIET [8] is a hierarchical load balancer for dispatching computational jobs over a distributed infrastructure, like a grid or a cloud. This use case is studied with regard to horizontal transformations applied to a component-based view. Its architecture is based on a set of agents: Master Agents (MA) manage pools of computational SErver Deamons (SED) via none, one or several strati of Layer Agents (LA). SEDs can achieve specialized computational services. Communications between agents are driven by the omniORB naming service (OMNI). MAs listen to client requests and dispatch them through the architecture to the best SED that can carry out the required service.

This application has been described using class diagrams [36], but, in addition to correction-granting issues, the fact that a LA can manage another LA could not be taken into consideration.

After lack of generality, a simplified architecture with a single MA and a single OMNI will be considered here. The main characteristics of the application are as follow:

1. While being deployed, each component records itself to the OMNI.
2. Each LA and each SED has a hierarchical superior (i.e., the parent node in the tree).
3. The MA and each LA manage from one \( \text{minSonsMA} / \text{minSonsLA} \) to ten \( \text{maxSonsMA} / \text{maxSonsMA} \) entities. Later in this paper, we will see that these conditions could be trivially extended to any number of minimum and maximum managed entities. Furthermore, from now on, an LA will be said to provide a certain service whenever at least of its child nodes does.
4. Due to hypothetical software restrictions and limited number of machines, the architecture is composed by at most one hundred agents. Once again, this arbitrary value could be expanded to any other.

Figure 1 offers a visual example of how a configuration of DIET may look like, with and without an OMNI. For obvious clarity concerns, the naming service will not be represented in future figures.

All instances of an architectural style are NOT created equal. At a given time, even though a configuration meets all the requirements of the application, another configuration may meet them in a “better way”. In particular, we consider the following criteria:

- the energy consumption,
- the robustness, i.e. the fault-tolerance w.r.t. the breakdown of a machine or a software component, and
- the quality of service.

We assume that the energy consumption depends only on the number of used machines and of the software components deployed on them.
<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA(d)</td>
<td>the set of LA of depth d</td>
</tr>
<tr>
<td>M(c)</td>
<td>the number of entities managed by the component c</td>
</tr>
<tr>
<td>maxSonsMA</td>
<td>the maximum number of entities managed respectively by the MA or a LA</td>
</tr>
<tr>
<td>minSonsMA</td>
<td>the minimum number of entities managed respectively by the MA or a LA</td>
</tr>
<tr>
<td>maxσ</td>
<td>threshold value of the balancing condition</td>
</tr>
</tbody>
</table>

The robustness, instead, is assessed based on three criteria that refers to the set of SEDs running the same service: (i) redundancy degree; (ii) location; (iii) balance within the hierarchical structure. For example, even if multiple SEDs are used for the same service (i.e., redundancy) it is important to allocate them on different machines (i.e., location) to reduce the vulnerability to hardware breakdowns. Similarly, spreading the SEDs far apart within the system tree helps improving the resiliency to LA failures (subject to the constraint that the LAs, not belonging to the same path from the MA to a SED, run on different machines).

Regarding quality of service, the load balance among the different vertices at the same depth in the tree is considered as criterion. Let $LA(d)$ be the set of LA of depth $d$, and $M(c)$ be the number of entities managed by the component $c$. An entity can be deployed and directly managed by a LA $\in LA(d)$ if it does not make the standard deviation of $\bigcup_{la \in LA(d)} M(la)$ become greater than a target threshold value, noted max$\sigma$.

**An interesting point here is that robustness and energy consumption are concurrent, in the sense that deploying more software components or using more machines will, while ameliorating the first, badly impact the second.**

To value these three objectives, it is crucial to keep track of some attributes of the software components:

1. the depth of each LA,
2. the number of entities managed by each component $c$ of type LA and MA: $M(c)$,
3. the set of services carried out by each SED and LA,
4. the machine on which each entity is deployed.

Notations introduced to describe DIET are summarized in Table 1.

### 3.2 Problem Statement

Herein we illustrate the main issues in modelling the DIET architecture using classical approaches:

1. **Interdependency of attributes**: the attributes of an entity $v$ may depend on attributes belonging to a set $S$ of other entities. In classical string grammars, attributes are classified as inherited or synthesized depending on whether the elements of $S$ are parents/siblings of $v$ in the parse tree or not, respectively. Conceptually, this means that the value of synthesized attributes cannot be known in the context where they first appear; further analysis through the application of production rules is necessary. In graph grammars, these rules traditionally symbolize the addition of software components. Similarly, graph attributes have to be handled in a very different way whether they only depend on attributes belonging to already existing entities or not. The first case can easily be addressed with attributes inheritance in graph grammars. For example, if we consider the depth of an LA as an attribute, we can derive it from the depth of the entity managing it. This does not apply in the second case. In fact, the set of services offered by an LA, for example, cannot be known in advance since they depends on children nodes that will be added or striped later on.

2. **Modification of an existing attribute**: as discussed in 2.2.2, the option of considering attributes as nodes has been discarded, because of scalability and efficiency issues. Consequently, graph rewriting rules cannot be used to modify attributes, and their modifications are therefore not possible within the framework of classical graph grammars. This concern becomes even more important if we consider the interdependency of attributes (point 1), since the attribute modifications should be propagated based on the dependencies.

3. **Conditional deployment**: whenever trying to deploy a LA or a SED, we should verify that it does not violate the maximal cardinality condition and the balancing condition. The first one could be handled in a structural, syntactic way, using negative application conditions, implying search for several homomorphisms and thus a high computational complexity. The balancing condition, however, is too complex to be reasonably managed using pattern matching. In addition, constraints have to take into account the fact attributes may be unknown.

4. **Evaluating a configuration**: soft and hard constraints can be used to reflect the appropriateness of a configuration. These constraints, closely related to the system and its components, exhibit similarities with attributes; they depend on attributes, are evolving, and components of the same type have similar requirements. Hence, their integrating in the model as any attributes is relevant. It is crucial to make the distinction between integrating constraints within the architectural style, building a constrained style, and restraining the architectural style. Existing graph-based approaches are restricted to the second case, where constraints are used to narrow the scope of correctness only. They are integrated to the model of
the style, e.g. to the type graph in [5], but not in the configurations themselves.

these approaches focus on attributes and structural transformation, but lightly treat the appropriateness of a configuration. Yet, constraints linked to such considerations are closely related to the architecture and its components. As a consequence, we wish, while constructing, deploying, or reconfiguring a configuration, to construct an easily evaluable set of constraints. These lasts model non-structural requirements and functional objectives of the application. As a result, their violation could be detected and automatically handled by a manager without requiring complex decision and without analysing the actual performances of the whole application.

It is crucial to make the distinction between integrating constraints within the architectural style, building a constrained style, and restraining the architectural style. In the second case, constraints are used to narrow the scope of correctness only. They are integrated to the model of the style, e.g. to the type graph in [5], but not in the configurations themselves.

These four points put under the spotlight the limits of classical graph-based formalism and the need for its expansion described in this paper.

4 Introducing Constraints and Mutators within Graph Rewriting Systems

4.1 Attributes, Constraints and Attributes Rewriting

4.1.1 Attributes

The proposed formalism conserves the simplicity and the computational efficiency of “listing” attributes as labels [31] while granting the possibility of applying any algebraic operator. An attribute is represented as a couple, whose first element is a constant, or a variable or the result of any function on other attributes as long as the signature of the function is respected. The second element is the domain of definition. The expression of the function may include quantifiers. We assume the canonic notation where \( Y^X \) is the set of function from \( X \) to \( Y \).

**Definition 1 (Attribute)** An attribute is a couple \( \text{Att} = (\text{Att}_A, \text{Att}_D) \) where
- \( \text{Att}_A \) is called value and is either
  - a variable,
  - a constant or
  - an expression, a regular combination of attributes if and only if \( \forall i \in \{1, n\}, D_{\text{Att}_D} \subseteq D_i \)

- \( \text{Att}_D \) is its domain of definition.

An attributed structure or system is a couple composed of the structure and a set of indexed attributes or sequence of attributes. The first member of an attribute will be noted within quotation marks if and only if it has a known value.

**Remark 1** In the approach presented herein, whenever considering an attributed structure itself composed by attributed elements, any attribute can be a regular combination of any others, not necessarily the attributes of the same structure or element.

For example, for an attributed structure \( (A, \text{ATT}_A) \) of attributed structures \( (B_i, \text{ATT}_A) \), let \( \text{ATT} = \bigcup_i \text{ATT}_A \cup \bigcup_i \text{ATT}_B \cup \bigcup_{(i,j)} \text{ATT}_C \), \( \forall \text{Att} \in \text{ATT}, \forall N \in \mathbb{N}, (\text{Att})_{k \in \{1, N\}} \in \text{ATT}^N \). Att can be a regular combination of \( (\text{Att})_{k \in \{1, N\}} \).

**Remark 2** Note that this notation is related to the approach considering many-sorted signatures and algebras [11], since elements are attributed over an implicit SIG-algebra. The sorts and operations of SIG are every defined domains of definition and functions from any combination of sorts to any sort, respectively.

For any attributed structure \( (\text{str}, \text{ATT}_{\text{str}}) \), let \( \text{ATT} \) be the set of attributes resulting of the union of any attribute defined in the framework of \( \text{str} \), which is not empty as at least \( \text{ATT}_{\text{str}} \subseteq \text{ATT} \). Let \( S = \{ s \mid \exists \text{Att}, (\text{Att}, s) \in \text{ATT} \} \) and \( \text{OP} = \{ f \mid \exists N \in \mathbb{N}, \exists (\text{Att})_{k \in \{0, N\}}, f \in D_{\text{ATT}_0}^{D_{\text{ATT}_1} \times \cdots \times D_{\text{ATT}_N}} \}. \text{str} \) is attributed over a \((\text{S},\text{OP})\)-algebra.

4.1.2 Constraints

Attributes are entirely aimed at providing information on an algebraic structure. Constraints can be seen as a specific kind of attributes taking values in a ternary predicate logic system.

**Definition 2 (Constraint)** A constraint Cons is an attribute \((\text{Cons}_C, \text{Cons}_D)\) with \(\text{Cons}_D = \{ \text{“true”}, \text{“false”}, \text{“unknown”} \}\).

Considering that constraints share the same domain of definition, it will be implicit from now on, so that a constraint \(\text{Cons} = (\text{Cons}_C, \text{Cons}_D)\) may be simply referred to as \(\text{Cons}\). In the following, the principles of Kleene’s strong logic [23, 24] are adopted, in particular its basic logic operations (\(\lor, \land, \neg, \Rightarrow\)) and the fact that the only truth value is “true”. The uniqueness of this truth value means that evaluations are pessimistic, i.e. “unknown” is supposed to be false.

**Remark 3** A constraint, as a regular combination of attributes, can be seen as a classical expression of a predicate ternary logic. Considering a ternary logic rather than a binary one
implies that unlike attributes, constraints can always be evaluated. The idea here is to associate any minimal logic expression that cannot be evaluated, due for example to an attribute implied in its expression being un-evaluable or variable, with “unknown”.

In order to lighten the notation, an attributed system with constraints, i.e. a triple composed by the system, a set of indexed attributes or sequence of attributes, and a set of indexed constraints or sequence of constraints, is called an AC-system or structure. Whenever defining an AC-system with AC-elements, rather than separating each sets of attributes (resp. constraints), a single family of sequence of attributes (resp. constraints) indexed by the sets of attributed (resp. constrained) elements is considered.

4.1.3 Attributes Rewriting

One of the issues evoked in section 3 is the fact that attributes are prone to evolve. A reconfiguration may thus impact the attributes of the system, the addition of a SED may for example modify the set of services carried out by some LAs. In the literature, classical string rewriting theory [30] tackles this issue by using mutators. A similar approach is adopted here.

**Definition 3 (A mutator on an AC-system)** A mutator on an AC-system is an arbitrary algorithm updating the value(s) of none, one or some attributes of the AC-system.

According to this definition, the scope of mutators remains limited to the modification of values. They can not be used neither to add or suppress an attribute nor to modify the domain of definition of an attribute.

4.2 Attributed Constrained Graph Modelling a Configuration

4.2.1 Definition

An AC-graph, modelling a software snapshot or configuration at a given time, consists in an AC-couple of two AC-sets of vertices and edges where an edge is a couple of vertices (source, destination). Following the commonly used conventions for standard graphical descriptions, one considers that vertices represent services or architectural components and edges correspond to their related interdependencies. Note that vertices, edges and the graph itself are AC-systems. According to remark 1, elements are attributed over the same algebra, i.e. an attribute or a constraint of the graph, a vertex or an edge may consist in any regular combination of attributes of the graph, any vertex and any edge. For any set S, the cardinality of S is represented as |S|.

**Definition 4 (AC-graph)** An AC-graph is defined by the system $G = (V, E, ATT, CONS)$ where

- $V$ and $E \subseteq V^2$ correspond respectively to the set of vertices and edges of the graph,
- $ATT$ (resp. $CONS$) is a family of sets $ATT_{el}$ (resp. $CONS_{el}$), where $el$ is an attributed (resp. constrained) element and can be a vertex, an edge or the graph itself. Consequently, $ATT$ (resp. $CONS$) is indexed by a subset of $V \cup E \cup \{G\}$. $ATT_{el}$ is a set of attributes (resp. constraints) of arbitrary length and containing the sequence of attributes $(ATT_{el}^i = (A_{el}^i, D_{el}^i))_{i \in [1, |ATT_{el}|]}$ (resp. $(CONS_{el}^i = (C_{el}^i, D_{el}^i))_{i \in [1, |CONS_{el}|]}$) of the element $el$.

The graph is partially attributed and constrained since $ATT$ and $CONS$ are indexed by a subset of $V \cup E \cup \{G\}$. In this way, an empty set of attributes or constraint is not required if an element is wished not to be attributed or constrained. Consequently, $0 \leq |ATT| \leq |V| + |E| + 1$ and $0 \leq |CONS| \leq |V| + |E| + 1$. Now that AC-graphs are defined, it is possible to represent a configuration of DIET as presented in section 3.

4.2.2 Modelling a Constrained Configuration of DIET

This subsection is dedicated to the definition of a DIET configuration. Concerns expressed in Sec. 3 are mapped into the theoretical concepts previously introduced in this Section. For sake of clarity, before formally introducing architectural styles, we show in Figure 2 what a DIET configuration would look like, once represented using an AC-graph.
Table 2 Notations used to describe a DIET configuration (see Fig. 2)

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mach</td>
<td>the set of available machines</td>
</tr>
<tr>
<td>Nat</td>
<td>the set of possible nature of a software component</td>
</tr>
<tr>
<td>Link</td>
<td>the set of possible relationships</td>
</tr>
<tr>
<td>S</td>
<td>the set of services that could be carried out by a SED</td>
</tr>
<tr>
<td>Serv</td>
<td>the power set of S</td>
</tr>
<tr>
<td>Red</td>
<td>the redundancy constraint</td>
</tr>
<tr>
<td>Loc</td>
<td>the localisation constraint</td>
</tr>
</tbody>
</table>

**Notations** are reported in Table 2.

- **Nat**, the set of possible nature of a software component, is equal to \{"OMNI", "MA", "LA", "SED"\}.
- **Link**, the set of possible relationships between entities, equals \{"ma2la", "ma2sed", "la2sed", "la2la", "registered"\}

**Red** and **Loc**, the redundancy and localisation constraints, are further described in the dedicated paragraph.

**Description of the configuration** At this time, the software is composed by eight components symbolized by eight vertices and their corresponding relations modelled by some edges, both attributed to reflect their properties and natures. A notable fact is that components of the same nature have the same number of attributes, theirs attributes being the one identified in Sect. 3. This is ensured by the definition of the rewriting system that will be presented later in this paper. Some components as well as the graph itself are constrained to reflect the concerns stated in the same section.

Constraints are represented within dotted frames, and related to their target elements by a dotted line, except for those linked to the graph itself.

**Attributes** The first attributes of each vertex states the nature of the modelled entity, in **Nat**. The configuration comprises a MA managing 2 entities and deployed on a machine noted \(m_1\), as represented by its second and third attribute, respectively.

Each LA possesses three more attributes, related to its depth, the number of entities it managed, the machine it is deployed on and its provided set of services. In the example, three LAs are deployed, respectively represented by \(v_2\), \(v_3\) and \(v_4\), of depth 1, 1 and 2, managing 2, 1 and 2 entities, and placed on machine \(m_2\), \(m_3\) and \(m_4\). The first one, \(v_3\), manages directly or indirectly SEDs providing the set of services \(s_1 \cup s_2 \cup s_3\), the second one, \(v_4\), provides \(s_4\) and \(s_5\), the third one, \(s_1 \cup s_2\).

Finally, four SEDs deployed on \(m_5\), \(m_6\), \(m_7\) and \(m_8\) carry out the services \(s_1\), \(s_2\), \(s_3\) and \(s_4\).

Note that machines and proposed services are represented by variable, and their actual value is not currently know.

**Constraints** The MA should not manage more than 10 entities, underlining a fundamental property of the architectural style. Load balancing is not represented since it is tackled by conditional deployment, as stated previously.

To cope with robustness, the graph is constrained by two clauses **Loc**(\(S,2\)) and **Red**(\(S,3\)), taking into account the need for redundancy and multiple locations over the offered services.

\(\forall s \in S, \forall x_i \in \mathcal{M}, \text{let the redundancy constraint } \text{Red}(\bar{S},x_i) \text{ be } \text{"There are at least } x_i \text{ SEDs carrying each service } s \text{ in } \bar{S}\". \text{Red}(\bar{S},x_i) = \forall s \in \bar{S}, \exists (v_i)_{i \in [1,x_i]} \in V^\nu, (\forall (i,j) \in [1,x_i]^2, i \neq j \Rightarrow v_i \neq v_j) \wedge (\forall k \in [1,x_i], \text{ATT}_{v_i} = \text{"SED" } \wedge \exists s \in \text{ATT}_{v_i}^2).\)

\(\forall s \in \bar{S}, \forall x_i \in \mathcal{M}, \text{let the location constraint } \text{Loc}(\bar{S},x_i) \text{ be } \text{"For each service in } \bar{S}, \text{there are at least } x_i \text{ different machines on which at least a SED carrying out the service } s \text{ is deployed".} \text{Loc}(\bar{S},x_i) = \forall s \in \bar{S}, \exists (v_i)_{i \in [1,x_i]} \in V^\nu, (\forall (i,j) \in [1,x_i]^2, i \neq j \Rightarrow (v_i \neq v_j \wedge \text{ATT}_{v_i}^1 \neq \text{ATT}_{v_j}^1)) \wedge (\forall k \in [1,x_i], \text{ATT}_{v_i}^1 = \text{"SED" } \wedge \exists s \in \text{ATT}_{v_i}^2).\)

This means that each service should be carried out by at least 3 SEDs located on at least two different machines.

In addition, a notion of location balance within sub-trees is introduced to re-enforce robustness. It is specified that a LA and a component managed by the same entity should not be deployed on the same machine or that they should have disjoint set of carried services. This constraint avoids, within a sub-tree, that SED providing similar services, and deployed in different location thanks to the clause **Loc**, are managed by entities deployed on the same machine. Hence the number of devices that have to breakdown in order for a service to be disrupted is increased.

**Formal definition** The graph in the Fig. 2 is defined as follow.

\(G = (V, \text{E, ATT, CONS})\) where
\(V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}\),
\(E = \{e_1 = (v_1, v_2), e_2 = (v_1, v_3), e_3 = (v_2, v_4), e_4 = (v_2, v_7), e_5 = (v_3, v_8), e_6 = (v_4, v_5), e_7 = (v_4, v_6)\}\),
\(\text{ATT} = \{\text{ATT}_G, \text{ATT}_{v_1}, \text{ATT}_{v_2}, \text{ATT}_{v_3}, \text{ATT}_{v_4}, \text{ATT}_{v_5}, \text{ATT}_{v_6}, \text{ATT}_{v_7}, \text{ATT}_{v_8}\}\).
\[ \text{Definition 5 (AC-graph homomorphism)} \] 
Two AC-graphs \( G = (V, E, \text{ATT}, \text{CONS}) \) and \( G' = (V', E', \text{ATT}', \text{CONS}') \) are homomorphic, noted \( G \rightarrow G' \), if and only if there is a graph-homomorphism \( h \) from \( (V, E) \) to \( (V', E') \) such as

1. \( \forall v \in V \ (\text{resp. } \forall e = (\overline{v}, \overline{v}) \in E), |\text{ATT}_v| = |\text{ATT}_{h(v)}| \) (resp. \( |\text{ATT}_e| = |\text{ATT}_{h(e)}| \))

2. \( \forall v \in V \ (\text{resp. } \forall e = (\overline{v}, \overline{v}) \in E), \forall i \in |\text{ATT}_v|, D_i = D_{h(v)} \)

3. The system of equations \( S = \{ A = A' \ | \ (\exists v \in V, \exists i \in |\text{ATT}_v|, A = A'_i \land A' = A'_{h(v)} \lor (\exists e = (\overline{v}, \overline{v}) \in E, \exists i \in |\text{ATT}_e|, A = A'_e \land A' = A'_{h(e)}) \} \) has at least one solution.

**Remark 4**

- Constraints do not impact on the definition of an homomorphism. It will be shown that they intervene in the rewriting process in a different way. Similarly, attributes on vertices and edges are the only one that are considered whereas attributes on the graph itself are not.
- Note that the existence of an AC-homomorphism is conditioned by the resolvability of a system of equations on attributes. As stated in the introduction, in attributed graphs \([18, 11]\), the existence of a morphism is also conditioned by equalities between attributes, potentially through the morphism between attributes spaces. However, this is often the only clause relying on attributes that impact the applicability of a graph rewriting rule.
- AC-graph isomorphism and AC-vertex identifications can be trivially deduced from the definition of AC-graph homomorphism.

Solving the system of equations \( S \) results in identifying the value of some attributes with some constants in their domains of definitions and/or with the value of some other attributes. Integrating the affection obtained by solving the systems refers to the update of the value of the attribute to reflect these identifications. For example, if \((x, y, (x, "2")) \in S^2\), meaning that \( x \) has been identified to the variable \( y \) and the constant \("2"\), integrating the affection obtained by solving \( S \) will lead to replacing each occurrence of \( x \) and \( y \) by \("2"\).

There exists a vast number of approaches handling graph rewriting based on attributed graphs \([18, 11]\). Their applicability depends on various factors, always including the existence of an homomorphism between an element of the graph rewriting rule and the graph to rewrite. Inspired by string grammar theory \([30]\), these factors are expanded herein to include the satisfaction of a set of constraints on attributes, namely the set of constraints of the AC-rewriting rule. This potentially empty set can be seen as a set of semantic predicates.

Applying a rewriting rule on a graph consists in suppressing a part of the graph and extending it by adding some
vertices and edges. In addition to classical modifications induced by the application of a rule, a set of actions is performed at the end of said application.

Virtually, any attributed graph rewriting formalism could be extended to include semantic predicates, constraints and mutators. In order to fix the idea, the classical double push out formalism defined in [33] has been chosen, alongside with the attribute management presented previously.

Definition 6 (AC-rewriting rule of AC-graph) An AC rewriting rule of an AC graph is a 5-tuple \((L, K, R, \text{ATT}, \text{CONS}, \text{ACT})\) where

- \(\text{ATT} = \text{ATT}_\text{rule} \cup \text{ATT}_L \cup \text{ATT}_R\) is a set of attributes, \(\text{ATT}_\text{rule}\) being the set of attributes of the graph rewriting rule itself,
- \(\text{CONS} = \text{CONS}_\text{rule} \cup \text{CONS}_R, K\) is a set of constraints, \(\text{CONS}_\text{rule}\) being the set of constraints of the graph rewriting rule itself and \(\text{CONS}_R, K\) the set of constraints of \(R, K\),
- \((L = (V_L, E_L), \text{ATT}_L, \emptyset)\) and \((R = (V_R, E_R), \text{ATT}_R, \text{CONS}_R, K)\) are AC-graphs,
- \(K = (V_K, E_K)\) is a sub-graph of both \(L\) and \(R\),
- \(\text{ACT}\) is a set of actions.

A rule is applicable on a AC-graph \(G\) if :
1. there is an AC-homomorphism \(h : (L, \text{ATT}_L, \text{CONS}_L) \to G\), implying in particular that the system of equations \(S = \{ A = A' | \exists v \in V_L, \exists i \in [1, |\text{ATT}_v|], A = A_i' \land A' = A_{h(v)}' \} \lor (\exists e = (i, i) \in E, \exists i \in [1, |\text{ATT}_e|], A = A_i', A' = A'_{h(i), h(i)}\}\) has at least a solution,
2. the application of the rule would not lead to the apparition of any dangling edge,
3. each \(\text{CONS} \in \text{CONS}_\text{rule}\) is evaluated to “true” by integrating the affectations obtained by solving \(S\) and by evaluating each elementary logic expression containing variable attributes to “unknown” as stated in remark 3.

Its application consists in :
1. erasing \(h(L \setminus K)\),
2. integrating the affectations obtained by solving \(S\) to the remaining graph,
3. adding an isomorph copy of \(R \setminus K\) integrating the affectations obtained by solving \(S\),
4. performing each action \(\text{Act} \in \text{ACT}\).

Efficiently representing evolving attributes : evaluation on demand or update on modification Note that, thanks to mutators, this formalism enforces several ways of considering and evaluating attributes or constraints. These lasts can be explicitly expressed as a regular combination of the graph attributes. However, this expression has to be calculated whenever its value is required. To avoid frequent evaluations, the attribute value can be stored and be updated whenever it has to be, using mutators. The choice between these two options rely on the relative complexities and frequencies of updates and evaluations. Besides, the adoption of a solution does not need to be final : one may evolve in run-time between the two of them to adapt to the evolution of said frequencies.

Inspired from Chomsky’s generative grammars [9], graph grammars are defined as a classical grammar or rewriting system, and formally characterize an architectural style.

Definition 7 (Graph Grammar) A graph grammar is defined by the 4-tuple \((AX, NT, T, P)\) where

- \(AX\) is the axiom, an AC-graph with a single vertex \(AX\),
- \(NT\) is a set of AC-vertices, called non-terminal term of the grammar,
- \(T\) is a set of AC-vertices terminal term, named terminal term of the grammar,
- \(P\) is the set of AC-rewriting rules, or production rules, belonging to the graph grammar.

Each vertex occurring in a graph rewriting rule in \(P\) or in a graph obtained by applying a sequence of productions \(\in P\) to the axiom is then isomorph to at least one arch-vertex in \(NT\) or \(T\).

Terminal terms define archetype of vertices with corresponding pattern of attributes and constraints. On the other hand, production rules grant constraint management and system updates. Terminal terms and productions guarantee that each component, at any time, of the system is correctly constrained and attributed according to its type.

Definition 8 (Instance belonging to the graph grammar) An instance belonging to the graph grammar \((AX, NT, T, P)\) is a graph obtained by applying a sequence of productions in \(P\) to \(AX\). If an instance does not contain any vertex isomorph to an arch-vertex from \(NT\) it is said to be consistent.

Correct-by-Construction Reconfigurations. Thanks to the very definition of compliance to an architectural style characterized by a graph grammar consistency preserving reconfigurations can be built from the productions rules. Correct by construction reconfigurations based on the generative aspect of graph grammars is one of theirs key advantages.

Consider any rewriting rule \(r\) whose application is equivalent to the application of a production or a sequence of productions of the grammar, noted \(p\) in this paragraph. Note that we can consider a single production even in the case of a sequence, through composition. It is immediate that \(p\) preserves consistency if its applicability conditions are equivalent or stronger than \(p\’\)'s ones, e.g. if \(r\) requires a larger pattern to be found meaning that \(L_r\) is a sub-graph of \(L_p\).
At first sight, it should be possible to terminate anything that has been started. Rules describing such processes can be obtained from the productions using graph rewriting rules’ property of reversibility. Let’s consider a consistent instance of a graph grammar, constructed by applying the sequence of productions rules \( \{p_i\}_{i \in \{1,...,N\}} \) to the axiom. Intuitively, if a reverse rule \( r \) is applicable, the relationship or entity it terminates has previously been started, meaning that there exists \( k \in \{1,...,N\} \) such as \( r \) is the reverse of \( p_k \). \( r \) preserves consistency if \( r \) and each rule in \( \{k, N\} \) are sequence independent.

### 4.4 Summary of the Proposed Contribution

In the previous Sub-Sections, a complete description of the proposed formalism has been detailed. Here, for sake of clarity, we highlight its pivotal features and advantages in a concise form.

- **Attributes** are enriched to cover their interdependencies and potentially unknown values. Their definition, rather than being restricted to predefined operators and dependencies, is based on the characterization of every admissible relationships.
- **Constraints** are defined as a special kind of attributes, so as to benefit from their evolution and dependencies mechanisms. Being elements of a ternary logic system, they cope with unknown attributes.
- **Graph rewriting rules** are expanded with the consideration of constraints and mutators. Firstly, constraints on the rule itself constitute semantic predicates that allow decision making in presence of unknown attributes. Constraints on the manipulated graph are tackled by a “merge” mechanism, allowing addition of new constraints on the fly. Secondly, mutators, adapted from classical string theory, manage efficiently and flexibly attribute modifications.

Accordingly, it is possible to extend graph grammar approaches in order to embrace these new features, capitalize their strengths, and enable the effective management of performance aware dynamic software architectures.

### 5 Exploitation of the New Formalism : DIET

#### Characterisation, Evaluation and Management

This Section illustrates the potential of the elaborated formalism by first describing DIET, taking into account each consideration introduced in Sect. 3. Then, the fitness of this description to appropriateness evaluation and performance aware management is demonstrated using concrete examples.

#### 5.1 Characterization of the Use Case.

This section is dedicated to the characterization of the DIET application described in Sect. 3 using the new formalism presented in this contribution. To this end, we design axioms, terminal terms, and production rules of the Graph Grammar that unambiguously define DIET. Also, we formally demonstrate the termination of the resulting grammar.

##### 5.1.1 Axiom

Considering the definition of graph rewriting rules and systems, instances of the such systems are graphs that inherit the attributes and constraints of the axiomatic graph. In the case of DIET, attributes and constraints shared by all possible software configurations are:

1. the largest number of entities that a LA may manage (the minimum being directly granted by production rules),
2. the largest number of entities that a MA may manage (idem),
3. the threshold value intervening in the balancing condition discussed in Sec. 3,
4. the maximum of total agents and
5. the current number of agents.

Common constraints, instead, refers to redundancy and location conditions each configuration has to satisfy.

Therefore, let \( AX_{DIET} \) be \( (VAX, ATTAX = ((\text{maxSonsLA}, N), (\text{maxSonsMA}, N), (\text{max}_{\sigma, \mathbb{R}}), \text{maxAgents}, N), (\text{curAgents}, N)), CONSAX = (\text{Loc}(S,2),\text{Red}(S,3))) \), where \( \text{curAgents} = 0 \) and, arbitrarily, \( \text{maxSonsMA} = \text{maxSonsLA} = 10 \) and \( \text{maxAgents} = 100 \).

Throughout this section, the graph on which production rules will be attempted to be applied to is noted \( G = (V, E, ATT, CONS) \). Attribute and constraint inheritance ensure that if \( G \) is an instance of the architectural style defined here, \( ATT_G = ATT_{AX} \) and \( CONS_G \subseteq CONS_{AX} \).

##### 5.1.2 Terminal Terms

These terms characterise types of AC-vertices, defining a pattern of attributes and constraints shared by vertices of the same kind.

The naming system itself is not constrained, and its attributes are limited to its nature and the machine it is deployed on. Therefore, let \( T_{Omni} \) be \( (\text{Omni}, ATT_{Omn} = (\text{“Omni”}, Nat), (m, Mach)), \emptyset) \).

Similarly, let \( T_{SED} = (\text{SeD}, ATT_{SED} = (\text{“SED”}, Nat), (s, Serv), (m, Mach)), \emptyset) \).
The MA shall not manage more than 10 entities. Accordingly, let \( T_{MA} \) be \((v_{MA}, ATT_{MA} = ("MA", \text{Nat}), (\text{Nsons, N}), (\text{m, Mach})), CONS_{MA} = ((\text{Nsons} < \text{A}_{\text{LA}}^{2}))\).

Finally, a LA and a component managed by the same entity should not be deployed on the same machine or they should have disjoint set of carried services. Let \( \hat{v} \) be the entity managing \( v \), \( \hat{v} \in V \), \( (\hat{v}, v) \in E \), and \( \text{sib}(v) = \{ v | (\hat{v}, v) \in E_{G} \} \) \( \setminus \{ v \} \) the set of components managed by \( \hat{v} \), excluding \( v \), i.e. the siblings of \( v \).

\( T_{LA} \) is \((v_{LA}, ATT_{LA} = ("LA", \text{Nat}), (\text{depth, N}), (\text{Nsons, N}), (\text{m, Mach}), (\text{s, Serv})), CONS_{LA} = (c(v_{LA})) \), where \( c(v) = \{ c(v) \} \in [1, |\text{sib}(v)|] \), \( \forall v \in \text{sib}(v), ! \exists i \in [1, |\text{sib}(v)|] \),

\[
A_{1}^{2} = \text{"LA"} \land c(v) = (A_{1}^{2} \neq A_{2}^{2} \lor (A_{1}^{2} \land A_{2}^{2} = 0)) \lor \ 
(A_{1}^{3} = \text{"SED"} \land c(v) = (A_{1}^{3} \neq A_{2}^{3} \lor (A_{1}^{3} \land A_{2}^{3} = 0)))
\]

5.1.3 Productions of the Grammar

Production rules of the graph grammar formalize the construction of its instances by defining when and how an entity may be deployed and the consequences of such a deployment.

The first rule \((p_{1})\) to define is the initialisation consuming the axiomatic vertex (Del). The naming service and the MA are deployed, as well as a non-terminal vertex granting that the MA manages at least an entity (Add). This vertex will be later on instantiated into a LA or a SED. Finally, the MA registers to the naming service and the current number of agents is updated accordingly.

Let \( p_{1} = (L_{p1}, K_{p1}, R_{p1}, \emptyset, \emptyset, \mu_{\text{registering}}(pv2), \mu_{\text{inc}}(G, 5, 3)) \), where \( \mu_{\text{registering}}(v) \) is the action of registering the object represented by the vertex \( v \) to the naming service. \( \mu_{\text{inc}}(e, i, 1) \), defined in Fig. 3, represent the incrementation of the \( i \)-th attribute of \( v \) by \( n \).

Graphical parts of the rewriting rules are illustrated here using the format L→K→R. This graphical representation is illustrated in Fig. 4, where \( L_{p1}, K_{p1}, R_{p1} \) and \( pv2 \) are defined.

\[
\mu_{\text{inc}}(e, i, n) \quad A_{i}^{2} \leftarrow A_{i}^{2} + n
\]

Fig. 3 \( \mu_{\text{inc}}(v, i, n) \). Incrementation of the \( i \)-th attribute of the element \( e \) by \( n \)

Productions rules \( p_{2} \) and \( p_{3} \) model the addition of a non-terminal vertex, managed respectively by the MA or a LA. This temporary vertex will later on be instantiated into a LA or a SED. To deploy a new entity, three condition should be met. This addition should respect (1) the balancing condition, (2) the maximal number of agents manageable by its superior and (3) the maximum number of total agents. The application of these productions leads to the incrementation of the numbers of total agents and of sons of the entity managing the added vertex.

Let \( p_{2} = (L_{p2}, K_{p2}, R_{p2}, \emptyset, A_{G}^{4} > A_{G}^{5}, (\mu_{\text{inc}}(pv2MA, 2, 1), \mu_{\text{inc}}(G, 5, 1))) \) and \( p_{3} = (L_{p3}, K_{p3}, R_{p3}, \emptyset, (\text{balancing}(pv2LA), \text{Nsons < ATT}_{G}, A_{G}^{5} > A_{G}^{5}, (\mu_{\text{inc}}(pv2LA, 3, 1), \mu_{\text{inc}}(G, 5, 1))) \), where \( L_{p2}, K_{p2}, R_{p2}, pv2LA, L_{p3}, K_{p3}, R_{p3}, \) and \( pv2MA \) are defined in Fig. 5. balancing(\( v \)) = \( \sigma((\text{A}_{LA}^{2})_{v} \cup \{ r \}, \text{A}_{LA}^{3}+1) < \text{A}_{LA}^{2} \), where \( \sigma(s) \) is the standard deviation of the sequence \( s \).

The instantiation of a temporary vertex managed by the MA or a LA into a SED is described by \( p_{4} \) and \( p_{5} \), respectively. After deploying the SED, it has to register to the naming service and, if it is managed by a LA, update the set of its carried out services. Let \( p_{4} = (L_{p4}, K_{p4}, R_{p4}, \emptyset, \emptyset, \mu_{\text{registering}}(pv4)) \) and \( p_{5} = (L_{p5}, K_{p5}, R_{p5}, \emptyset, (\mu_{\text{registering}}(pv4), \mu_{\text{updateServ}}(pv2, pv2, 4), \mu_{\text{inc}}(G, 5, 1))) \), where \( L_{p4}, K_{p4}, R_{p4}, L_{p5}, K_{p5}, R_{p5}, \) pv2, and \( pv4 \) are defined in Fig. 7. \( \mu_{\text{updateServ}}(v, \hat{v}, \text{ind}) \), described in Fig. 6, impact a change in \( A_{piv}^{\text{ind}} \), the set of carried out services by \( \hat{v} \), on \( v \), the component managing \( \hat{v} \), by updating the set of services it proposes. This update is conducted only if \( v \) is a LA, and, if there is indeed a change, it is propagated to the entity managing \( v \).

The two last productions of the grammar, \( p_{6} \) and \( p_{7} \), describe the instantiation of non-terminal term into a LA managed respectively by the MA and a LA. Since a LA has to manage at least one entity, such an instantiation can be conducted only if an entity can be later on deployed without exceeding the maximum number of agents. Let \( p_{6} = (L_{p6}, K_{p6}, R_{p6}, \emptyset, A_{G}^{4} > A_{G}^{5}, (\mu_{\text{registering}}(pv4), \mu_{\text{inc}}(G, 5, 1))) \) and \( p_{7} = (L_{p7}, K_{p7}, R_{p7}, \emptyset, A_{G}^{5} > A_{G}^{5}, (\mu_{\text{registering}}(pv4), \mu_{\text{inc}}(G, 5, 1))) \), where \( L_{p6}, K_{p6}, R_{p6}, L_{p7}, K_{p7}, R_{p7}, \) and \( pv4 \) are defined in Fig. 8.
5.1.4 The Constrained Attributed Graph Grammar Characterizing DIET

Considering the sets introduced in this section, GRS\textsubscript{DIET}, the graph rewriting system formally characterizing DIET, the use case introduced in Sect. 3, is defined as GRS\textsubscript{DIET} = (AX\textsubscript{DIET}, NT\textsubscript{DIET}, TD\textsubscript{DIET}, PD\textsubscript{DIET}), where NT\textsubscript{DIET} = \{V\textsubscript{temp}, ATT\textsubscript{temp} = \{“temp”\}, CONS\textsubscript{temp} = \emptyset\}, 
TD\textsubscript{DIET} = \{T\textsubscript{Omni}, T\textsubscript{MA}, T\textsubscript{LA}, T\textsubscript{SED}\}, and 
PD\textsubscript{DIET} = \{p\textsubscript{1}, p\textsubscript{2}, p\textsubscript{3}, p\textsubscript{4}, p\textsubscript{5}, p\textsubscript{6}, p\textsubscript{7}\}.

Note that the limitation of entities that can be managed by the MA or a LA are not tackled in the same way. A constraint reflecting this restriction is added on the MA, whereas the satisfaction of this limitation is granted for the LAs by a semantic predicate. Said predicate restricts the applicability of p\textsubscript{3} by imposing, before making a LA manage a new component, that said LA as not reach the limit of component it can manage. Since p\textsubscript{3} is the only production of the grammar increasing the number of entities managed by a LA, this limit can not be overpassed. A recap of the mapping between the concerns expressed in Sec. 3 and formal concept is presented in Table 3.
Table 3 Informal considerations and their formal translation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Formal expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSMA</td>
<td>Nsons &lt; A$^2_{\text{LA}}$</td>
<td>The MA and LAs are deployed alongside with respectively $\text{minSonsMA}$ and $\text{minSonsLA}$ temporary vertices</td>
</tr>
<tr>
<td>CONS$^2_{\text{pi}}$</td>
<td>Nsons &lt; $\text{ATT}_i^3$</td>
<td>The MA is constrained not to exceed $\text{maxSonsMA}$</td>
</tr>
<tr>
<td>CONS$^1_{\text{LA}}$, CONS$^2_{\text{LA}}$, CONS$^3_{\text{LA}}$, CONS$^4_{\text{LA}}$</td>
<td>$A^0_{\text{LA}} &gt; A^1_{\text{LA}}$</td>
<td>Any transformation eventually implying the deployment of a new component can be applied only if the system is currently composed by less than $\text{maxAgents}$ entities</td>
</tr>
<tr>
<td>CONS$^1_{\text{pi}}$ = balancing(v)</td>
<td>$\sigma((A^3_{\text{LA}})<em>{u \in \text{LA}(A^2</em>{\text{pi}})(v) \cup A^1_{\text{pi}} + 1}) &lt; A^0_{\text{pi}}$</td>
<td>Incrementing the number of entities managed by v respecting the balance condition</td>
</tr>
<tr>
<td>sib(v) = balancing(v)</td>
<td>${ \forall i \in E_{\text{pi}} } \setminus {v}$</td>
<td>Siblings of v</td>
</tr>
<tr>
<td>CONS$_{\text{LA}}$ = c(v)</td>
<td>$(c(v)i)_{i \leq \lceil</td>
<td>\text{sib}(v)</td>
</tr>
<tr>
<td>CONS$<em>{\text{LA}}$ = Red(S,X) = \forall s \in \hat{S}, \exists (v_i)</em>{i \in [1,</td>
<td>x_i</td>
<td>]} \in V^x, (\forall (i,j) \in [1,</td>
</tr>
<tr>
<td>CONS$<em>{\text{LA}}$ = Loc(S,2) = \forall s \in \hat{S}, \exists (v_i)</em>{i \in [1,</td>
<td>x_i</td>
<td>]} \in V^x, (\forall (i,j) \in [1,</td>
</tr>
</tbody>
</table>

Let’s consider the following system of tokens:
- Token A : G4 - G5, the number of agents that still can be deployed.
- Token B : the number of temporary vertices in the graph.

Applying $p_2$, $p_3$, $p_6$ or $p_7$ decrease the number of tokens in A by 1, whereas $p_1$ requires 3. Hence,  

$$3 \ast \text{Occ}(p_1) + \text{Occ}(p_2) + \text{Occ}(p_3) + \text{Occ}(p_6) + \text{Occ}(p_7) \leq \text{maxAgents}. \quad (2)$$

The application of $p_4$ or $p_5$ consumes 1 token B, whose number is increased when applying $p_1$, $p_2$ or $p_3$. Hence,  

$$\text{Occ}(p_4) + \text{Occ}(p_5) \leq \text{Occ}(p_1) + \text{Occ}(p_2) + \text{Occ}(p_3). \quad (3)$$

Since $p_1$ consumes the axiom, it is obvious that  

$$\text{Occ}(p_1) = 1. \quad (4)$$

Equation (2) thus becomes:  

$$\text{Occ}(p_2) + \text{Occ}(p_3) + \text{Occ}(p_6) + \text{Occ}(p_7) \leq \text{maxAgents} - 3. \quad (5)$$

By definition, $\forall p \in \text{P_{DIET}}, \text{Occ}(p) \geq 0$. Accordingly, equations (3), (4) and (5) give:

$$\text{Occ}(p_2) + \text{Occ}(p_3) + \text{Occ}(p_6) + \text{Occ}(p_7) \leq \text{maxAgents} - 3.$$

**Termination of the grammar.** A generative grammar is said to be terminating if there can not be an infinite sequence of its production rules. Theoretically, this property ensure that the set of instances of the grammar is finite and that its exploration or the construction of an instance can be represented by a terminating algorithm. Practically, this property is consistent with the finiteness of the available resources, like machines and their computing powers.

**Theorem 1 GRS$_{\text{DIET}}$ is terminating.**

**Proof.** Let $S$ be a non-empty sequence of elements in $\text{P}_{\text{DIET}}$ and $|S|$ its size. Let’s prove that $\exists N \in \mathbb{N}, |S| \leq N$.

For $p \in \text{P}_{\text{DIET}}$, let $\text{Occ}(p)$ be the number of occurrences of $p$ in $S_p$. Accordingly, 

$$|S| = \sum_{p \in \text{P}_{\text{DIET}}} \text{Occ}(p). \quad (1)$$
\[ \text{Occ}(p_4) + \text{Occ}(p_5) \leq \text{maxAgents} - 2. \]  

(6)

According to equation (1),

\[ |S| = \text{Occ}(p_1) + (\text{Occ}(p_2) + \text{Occ}(p_3) + \text{Occ}(p_6) + \text{Occ}(p_7)) + (\text{Occ}(p_4) + \text{Occ}(p_5)). \]

Thanks to equations (4), (5) and (6), this translates to

\[ |S| \leq 1 + (\text{maxAgents} - 3) + (\text{maxAgents} - 2). \]

Finally, \( |S| \leq 2 \times \text{maxAgents} - 4. \) With \text{maxAgents} = 100, we have \( |S| \leq 196. \)

QED.

5.2 Appropriateness Evaluation

To enable the evaluation of the appropriateness of a configuration in DIET, herein, constraints are assigned a, potentially infinite, weight. The cost of a configuration is calculated as the sum of the costs of its energy consumption and of the violated constraints. The violation of a constraint in a configuration implies that every defined criteria is not respected. In this case, the configuration is not robust enough and its cost is therefore increased depending on the weight of the violated constraint. A configuration of infinite weight is considered incorrect, so that strong constraints are still enforced.

Notations Let \( \xi \) be the function of evaluation: \( \forall \ c \in \text{CONS} \), \( \forall \ c \in \text{cons} \), \( \xi(c) = 1 \) if \( c \) is “true” and 0 else. Let \text{UsedMach} be the set of used machines.

Energy Consumption In Sec. 3, it has been assumed that energy consumption depends on the used machines and the number of deployed components only. In the following, this relation is supposed linear, and weighted by \( \lambda_{\text{mach}} \) and \( \lambda_{\text{entity}} \) respectively for used machine and deployed component. For an easier evaluation, the number of used machines and deployed components can be added as attributes of the graph and updated whenever necessary, i.e. when applying production \( p_1, p_4, p_5, p_6 \) or \( p_7 \). The energy consumed by a configuration is then: \( \lambda_{\text{mach}} \cdot |\text{UsedMach}| + \lambda_{\text{entity}} \cdot |V|. \)

Constraint violations It is clear that the constraint reflecting the limitation on the number of entities managed by the MA should not be violated and therefore has an infinite weight. Constraints reflecting the robustness of the system are, however “soft” and are given arbitrary finite weight. The cost of violating the constraint stating that “a LA and a component managed by the same entity should not be deployed on the same machine or they should have disjoint set of carried services”, \( c(v) \), is weighted by the depth of the LA. Redundancy and location constraints are weighted respectively by \( \lambda_R \) and \( \lambda_L \).

The cost related to the violation of constraints is:

\[ \lambda_L \cdot \xi(\text{CONS}_L^1) + \lambda_R \cdot \xi(\text{CONS}_R^1) + \lambda_{\text{MA}} \sum_{\text{ma} \in V, \text{A}_1 = \text{MA}} \xi(\text{CONS}_1^1) + \sum_{\text{la} \in V, \text{A}_2 = \text{LA}} \lambda_L \cdot (A^2 \cdot \xi(\text{CONS}_la^1)). \]

Part of the configuration illustrated in Fig. 2 is arbirarily instantiated to be totally evaluable and presented in Fig. 9. \( S, \) the set of services that may be carried out by a SED, is \( \{S_1, S_2, S_3\} \).

This configuration does not meet the redundancy constraint, since there are only two SEDs that can carry out the services \( S_1 \) and \( S_2 \). Hence, its cost is equal to its energy consumption plus the cost of violating said constraint: \( 3\lambda_{\text{mach}} + 7\lambda_{\text{entity}} + \lambda_R \).

This natural and immediate way to deal with soft and hard requirements derives from the new formalism proposed in this paper. In fact, the model has been explicitly conceived to embed constraints, performance indicator and their admissible soft bounds by means of attributes. In this way, once the software architecture properly described, its appropriateness can be easily evaluated on a dynamical basis.

5.3 Performance Aware Management

Semantic predicates and restrictions on rules’ applicability grant more flexibility on transformations, allowing also to face specific aims on the fly. These restraints are not fixed...
and can arise, evolve or disappear, as the motivation of the reconfiguration change. Let’s consider the DIET configuration previously evaluated (see Fig. 9). In this case, we can suppose that a SED is to be deployed to meet the redundancy constraint and improve the quality of the configuration.

The first thing to do is to apply $p_2$, and by doing so choosing the component that will manage the SED. To find an optimum solution, one should consider each possibility, i.e. apply $p_2$ on $v_2$, $v_3$ or $v_4$, find in each case an optimal solution, and then compare the costs of each solutions.

Let’s assume that $p_2$ has been applied on $v_2$, and let’s find an optimal solution in this case. The temporary vertex is to be instantiated into a SED using the production $p_5$. Since the motivation of this reconfiguration is to meet the redundancy constraint, $p_5$ should be restricted in order for its application to be relevant.

Firstly, the deployed SED should be able to provide the services $S_1$ and $S_2$. Assuming the notation introduced when defining $p_5$, see Fig.7, $\{S_1, S_2\} \subseteq A_{pv4}^2$.

Furthermore, the constraint (“$M_2$! = $A_{pv4}^3$”) ∨ (“$\{S_1,S_2,S_3\}$” ∩ $A_{pv4}^3 = \emptyset”) will appear on $v_3$. In order for this constraint to be met, the transformation should verify that $A_{pv4}^3 \neq “M_2”$.

Besides, for energy consumption reason and so as not to use a new machine, it is imposed that $A_{pv4}^3 \in \{M_1, M_3\}$.

Figure 10 presents a possible optimal -according to the style-defined constraints- result of such a reconfiguration. Every constraints are met and the cost of the configuration is now limited to its energy consumption, $3\lambda_{mach} + 8\lambda_{entity}$. Hence, this evolution is relevant iff $\lambda_{entity} < \lambda_R$

6 Conclusion

Dynamic software architectures enable adaptation in evolving distributed systems, focusing on two particular aspects which are usually considered separately: correctness and performance. Graph and graph rewriting based methodologies are appropriate for designing correct-by-construction reconfigurations of dynamic systems, effectively guaranteeing their consistency by requiring little or no verification in run-time. Their genericness allows the representation of a vast range of systems in different fields, with various views and dynamical aspects. Within the field of software architecture in particular, graph-based approaches may be used to represent, for example, architecture that are component-based [12], service-based [5], event-oriented, or even human applications [31], with both horizontal or vertical transformations, i.e. reconfigurations or refinements.

With reference to DIET, an industrial application contributing in federating and managing hybrid HPC environment, this article first show that currently available graph based methods however exhibit limitations in handling performance related properties of a system. Then, to conciliate correctness and performance, an extension of graph grammars is proposed so as to lift the highlighted restrictions. The pivotal features of this new formalism are: mutators, admissible relationships specification, and constraint oriented encoding. The firsts are introduced within graph rewriting rules as a lightweight approach to attribute modifications. On the other hand, attribute interdependencies are expressed through algebraic operators that allow to characterize admissible relationships. Finally, to ease application management operations, the appropriateness of a configuration in accordance to secondary objectives is reflected by constraints. Noticeably, to cope also with unknown attributes, constraints are defined as elements of a ternary logic systems.

The application of the resulting formalism to the specification of DIET demonstrates its fitness for performance aware management. First, it allows to efficiently assess characteristics of the system by combining evaluation on demand and/or update on modification. Secondly, it is shown using concrete examples that the model allows to quickly grasp the appropriateness of a configuration, identify objectives that can be ameliorated, and component implying constraints violation. Third, we successfully investigated the potential of the new formalism to guarantee theoretical properties of a grammar like its termination.
Mechanisms to take advantage of this new model are currently being integrated within FRAMESELF [1], a multi-model framework for self-management of distributed systems. Also, ongoing research is exploring the suitability of the proposed formalism to time-constraints. Another interesting on-going development is the consideration of infinite logic systems. These last provide a way of transforming qualitative properties into quantitative one, by linking, for example, robustness to failure probability.

References

24. Kleene SC (1952) Introduction to metamathematics. Bibliotheca mathematica (vol. 1), North-Holland, Amsterdam
27. Lun L, Chi X (2010) Relationship between testing criteria for architecture configuration testing based on Wright specification. In: 2010 International Conference on Computational Intelligence and Software Engineering (CiSE), pp 1–4

Vitae

Cédric Eichler is a PhD candidate at LAAS-CNRS and IRIT. After obtaining a Certified Engineer Degree in computer science and applied mathematics, he worked on formal methods for the specification of distributed systems. He is involved in the development of autonomous policies based on correct by construction reconfiguration for the national ANR SOP project.

Thierry Monteil is an associate professor in computer science at INSA of Toulouse and researcher at LAAS-CNRS. He has a Doctorate in parallel computing and Certified Engineer Degree in computer science and applied mathematics. He works on parallel computing (LANDA parallel environment), Grid resources management (AROMA project), Internet of things (OM2M project), computer and network modeling, autonomous policies to improve performance on distributed applications and parallelization of large electromagnetic simulation.

Patricia Stolf is an assistant professor at the Toulouse University (France). She obtained a PhD in 2004 in the LAAS-CNRS laboratory (Toulouse-France) on Tasks scheduling on clusters for remote services with quality of service. She is currently working in the IRIT laboratory on the field of distributed algorithms and autonomic computing in large scale distributed systems like grid and clouds. She studies resources management, load-balancing, energy aware autonomic systems and energy and thermal-aware task scheduling. She is involved in different research projects : in the ACTION COST IC0804 Energy Efficiency in Large Scale Distributed Systems, in the European CoolEmAll project and in the national ANR SOP project.
Khalil Drira is Directeur de Recherche, a full-time research position at LAAS-CNRS. Khalil Drira's research interests include formal design, implementation, testing and provisioning of distributed communicating systems and cooperative networked services, model-based analysis and design of correctness properties including testability, robustness, adaptability and reconfiguration. He is author of more than 250 regular and invited papers in international conferences and journals. He is or has been initiator of different national and international projects and collaborations in the field of networked services and distributed and communicating systems. Khalil Drira is member of the programme committees of international and national conferences. He is editor and member of the editorial board of different international journals in the field of software architecture and communicating and distributed systems.